修士論文

スケーラブルかつ高精度な Spectre 脆弱性の検出手法

吉田 昂太

23M30767

東京科学大学
情報理工学院
情報工学系
情報工学コース

指導教員　権藤 克彦

2025 年 1 月

概要

多くの CPU は投機的実行を悪用する Spectre 攻撃に対して脆弱である。これに対処するため、プログラム内の Spectre 攻撃に脆弱なコード片（ガジェット）を特定し、部分的に投機的実行を抑制する方法が提案されている。既存の記号実行を用いたガジェットの検出手法は、通常の実行パスに加え、投機的実行パスも探索する必要があるため、状態空間が増大し、大規模なプログラムに対しスケールしない問題がある。本論文では、ガジェットの検出可能性の低い投機的状態の探索を回避することで、記号実行のスケーラビリティを向上させる手法を提案する。さらに、探索されなかった投機的状態はファジングにより探索し、スケーラビリティと精度の両立を図る。

目次

図目次

表目次

第 1 章

はじめに

第 2 章

背景

2.1 投機実行

現代の CPU は 1 つの命令を、命令フェッチ、デコード、実行などの複数のステージに分割して実行するパイプライン方式を採用している。この方式により、前の命令が全ての処理を終えることを待たずに、次の命令の処理を開始できる。このように複数の命令を並行して処理することで CPU はスループットを向上させている。

しかし、次に実行すべき命令が前の命令の実行結果に依存している場合、CPU は次にどの命令を実行するべきか判断できなくなり、命令の処理を一時的に停止させる必要がある。このような状況は制御ハザードと呼ばれ、CPU のパフォーマンスに大きな影響を与える可能性がある。このようなハザードによる命令の実行の停止 (Stall) を回避するため、現代の CP は分岐命令に遭遇した場合、過去の実行結果に基づいて分岐先を予測することで、後続の命令を投機的に実行する。分岐予測による投機実行がどのように行われるかを図 2.1 に示す。図 2.1 に示すように、分岐予測に成功した場合、CPU は stall を回避し、効率的に命令を処理できる。一方、予測が失敗した場合は、投機実行した命令の結果を破棄し、正しい分岐方向から命令の実行を再開する必要がある。分岐先の予測には分岐予測器が用いられ、分岐命令のアドレスごとに過去の分岐方向を記録し、その情報から分岐先の予測を行う。

投機的に実行された命令と実行結果は CPU 内部の Reorder Buffer (ROB) という機構で管理される。ROB は、実行された命令が依存する他の命令の完了を待ちつつ、命令の実行順序を維持する役割を持つ。そのため、投機的に実行できる命令数は ROB の大きさに制限されており、一般的には 200 命令程度のマイクロオペレーション (μOP) に制限される。分岐先が確定し予測が正しかった場合は、投機的実行の結果をレジスタやメモリなどのハードウェア状態に反映させる。一方、分岐先が誤っていた場合は ROB 内の投機的実行の結果を全て破棄し、誤った予測が行われた時点のアーキテクチャ状態までロールバックされる。

2.2 Side-Channel Attack

コンピュータのシステムにおいて、channel とは情報を送信する可能性のある媒体のことを言う。channel には大きく分けて legitimate channel と incidental channel の 2 種類が存在する [1]。legitimate channel はシステムの設計者が情報送信用に意図したチャネルであり、イーサネット、共有メモリ、IPC ソケットなどがある。逆に incidental channel は 偶発的に設計されたチャネルであり、リソースの競合、CPU キャッシュの状態、電力消費の変化などがある。更に、セキュリティ脅威モデルのコンテキストにおいて incidental channel は covert channel と side channel の 2 種類に分類される。covert channel は悪意のある送信者と受信者が意図的に情報の伝達を行うために利用される channel である。一方で side channel は、送信者は受信者に情報を伝達することを意図しておらず、情報が悪意の

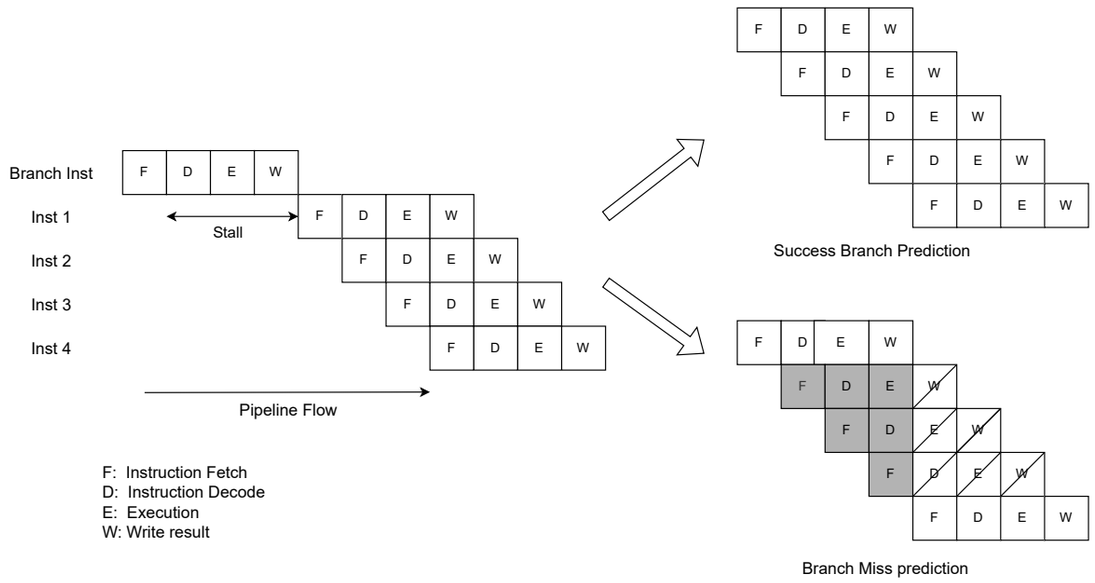

図 2.1: 分岐予測による投機実行

ある受信者に伝達 (つまり漏洩) される際に利用される channel であり、このような channel を悪用する攻撃手法を Side-Channel Attack という。

side channel は 情報を伝達する方法に基づいて、Timing-based channel、Accessbased channel、または Trace-based channel に分類できる [24]。Timing-based channel は、様々な操作のタイミングを利用することで被害者の情報を推測する。例えば、攻撃者は様々な入力を暗号化または復号化し、その時の実行時間の差分を計測し分析することで、暗号鍵に関する情報が明らかになる可能性がある [22]。Accessbased channels は メモリやキャッシュなどの共有リソースに直接アクセスすることで、被害者のプロセスの情報を推測する。例えば、キャッシュがプロセス間で共有されていることを利用し、特定のキャッシュラインへのアクセス時間を計測することで、被害者のプロセスの動作を推測することが出来る [16]。Trace-based channels は デバイスの消費電力や電磁放射などのプログラム実行時の詳細な情報を計測することで情報の漏洩を試みる。例えば、攻撃者は暗号化中のデバイスの消費電力を計測し分析することで、暗号化に関する情報を収集する [3]。

以降の節では、近年の CPU のデータキャッシュの基本的な構造と、それを side channel として利用する代表的な攻撃手法である Prime+Probe [25] と Flush+Reload [31] について説明する。これらの攻撃は本研究の対象である、Spectre 攻撃においても頻繁に利用される攻撃手法であるため、以降の節で詳しく説明する。

2.2.1 キャッシュ構造

キャッシュを side channel として利用する攻撃手法を紹介する前に、近年の CPU におけるキャッシュ構造について説明する。近年の CPU では、プロセッサとメインメモリの性能差に効率的に対処するため、複数の性能が異なるキャッシュメモリを階層的に配置する設計が採用されている。このキャッシュ階層は、CPU に近い順に L1 キャッシュ、L2 キャッシュと続き、最も遠いキャッシュは LLC (Last Level Cache) と呼ばれる。これらのキャッシュは CPU に近いキャッシュほど高速だが容量が小さく、遠いほど容量は大きいが、低速であるという特徴を持つ。一般的に L1 キャッシュと L2 キャッシュは各コア専用で、LLC は複数のコアで共有されているため、LLC は Side-Channel Attack の対象として適している。

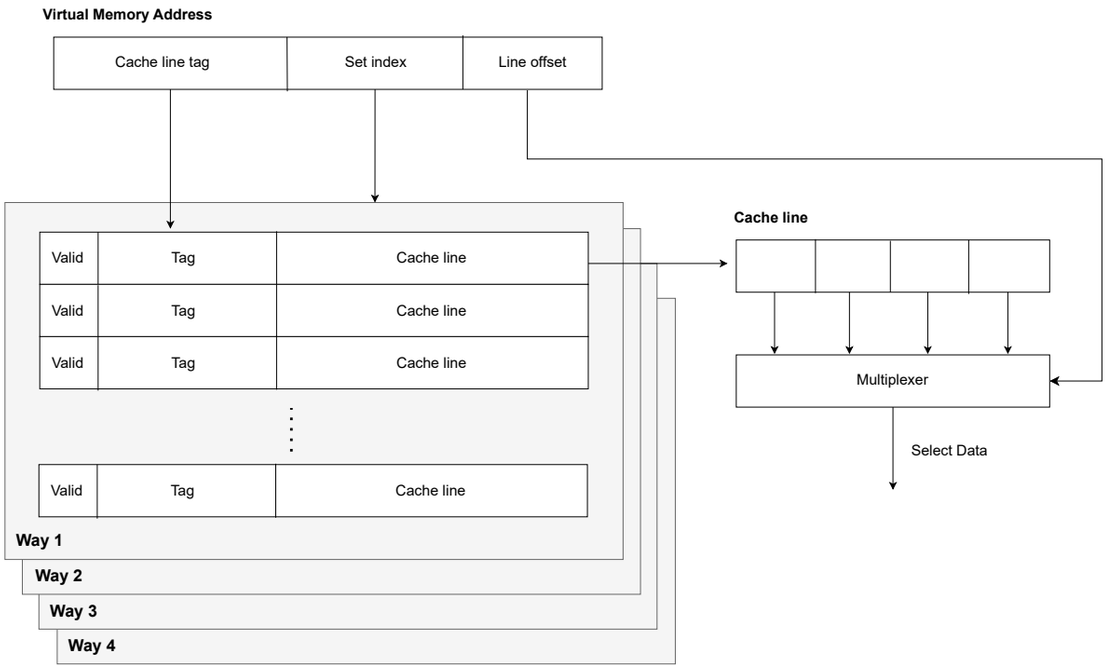

図 2.2: 仮想アドレスによるキャッシュの参照

キャッシュは複数のキャッシュセットで構成され、各セットには複数のキャッシュラインが含まれている。このキャッシュ構造をセットアソシエイティブ型と呼び、各キャッシュセット内のキャッシュライン数を連想度 (way) という。連想度が増加すると、キャッシュの競合性ミスが減少するが、比較器やキャッシュの設計コストが増加するため、バランスの取れた設計が求められる (TODO: intel CPU の way 数は?)。

仮想メモリアドレスは、以下の 3 つの要素で構成される。

- Cache set index: キャッシュ内でデータがどのセットにマップされるかを決定する。
- Cache line tag: キャッシュセット内でデータを識別するタグ。
- Cache line tag: キャッシュライン内でどのデータワードが対象であるかを識別するタグ。

キャッシュは仮想アドレスの一部をインデックスとして使用し、各メモリアドレスを特定のキャッシュセットにマップする。そして、セット内の任意のキャッシュラインにデータを格納する。この際、マップ先のキャッシュセットがすでに埋まっている場合は、キャッシュ置換ポリシーに基づいて、どのキャッシュラインを削除するかが決定される。一般的に採用される手法は、Least Recently Used (LRU) や Least Frequently Used (LFU) といったポリシーである。仮想メモリアドレスによる 4way セットアソシエイティブ型キャッシュの参照の概要を図 2.2 に示す。

2.2.2 Prime+Probe

Prime+Probe [25] は データキャッシュを side channel として利用し、キャッシュ内の被害者のデータアクセスパターンから秘密情報を抽出する攻撃手法である。攻撃の概要について図 2.3 に示す。まず、攻撃者は、攻撃者プロセスにおけるコードブロックと被害者プロセスにおける機密性の高いコードブロックの両方がマップされるキャッシュセットを見つける。先述の通り、一部のキャッシュ階層は複数コア間で共有されているため、攻撃者と被害者のプロセスが同一コアにスケジュールされていなくても攻撃することは可能である [16]。そして、攻撃者プロセスを一定時間実行することで、被害者と競合

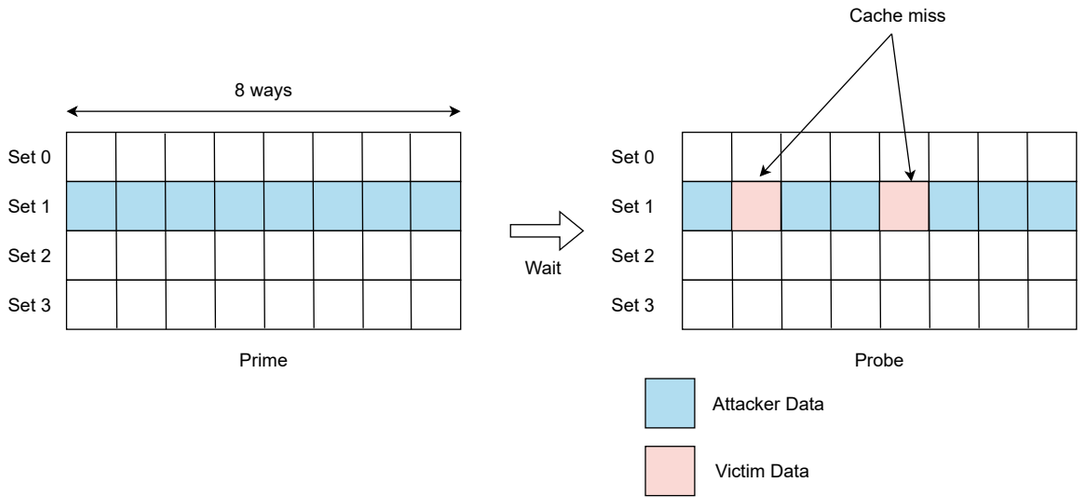

図 2.3: Prime+Probe の概要

するキャッシュセットを自身のデータで埋める (Prime)。その後、攻撃者は一定時間待機し、被害者プロセスが実行されるのを待つ。これにより、競合したキャッシュセットの一部は被害者のデータによって上書きされる可能性がある。その後、攻撃者は Prime フェーズで埋めたデータに再度アクセスし、アクセス時間を計測する (Probe)。アクセス時間を計測することで、攻撃者のデータがキャッシュ上に存在するかがわかるため、待機時間の間にどのキャッシュラインが被害者プロセスによってアクセスされたがわかる。つまり、攻撃者は特定のキャッシュセットにおける被害者プロセスのメモリアクセスパターンを把握することが可能となる。

Prime+Probe は後述する Flush+Reload と比較して、フラッシュ命令や共有メモリを必要としないためより汎用的である。しかし、対象とするキャッシュが LLC のように大容量の場合、キャッシュセットを攻撃者のデータで埋めるまでに時間がかかるという欠点がある。

2.2.3 Flush+Reload

Flush+Reload は Prime+Probe と同様に、データキャッシュを利用する Side-Channel 攻撃であるが、前提条件として被害者と攻撃者のプロセス間で特定のメモリ領域を共有している必要がある。プロセス間でメモリを共有するシナリオとして、プロセス間通信のための利用される場合や、メモリフットプリントの削減のために、ハイパーバイザが仮想マシン間で同一内容のメモリページを結合する際に利用される [31]。

まず、攻撃者は共有されたメモリ領域の中から観測対象とするアドレスを選択し、clflush 命令などを使用して、選択したアドレスのキャッシュラインをキャッシュ階層全体から排除する (Flush)。その後、攻撃者は一定時間待機し、被害者プロセスが観測対象のアドレスにアクセスするのを待つ。その後、攻撃者は観測対象のアドレスに再度アクセスし、アクセス時間を計測する (Reload)。待機中に被害者プロセスが観測対象のアドレスにアクセスした場合、そのキャッシュラインはキャッシュで利用可能になり、アクセス時間は短くなる。一方、被害者プロセスが観測対象のアドレスにアクセスしていない場合、キャッシュラインをメモリから取得する必要があるため、アクセス時間は長くなる。このようにして、攻撃者は待機中に被害者プロセスが観測対象のアドレスにアクセスしたかどうかを把握することが可能となる。

Flush+Reload は攻撃のために前提条件が必要だが、Prime+Probe と比較して特定のアドレスのキャッシュラインをフラッシュすれば良いだけので、攻撃にかかる時間が短いという利点がある。

```
1 i = input();
2 if (i < array1_size) { // VB: Victim branch
3   secret = array1[i]; // RS: Read Secret
4   tmp &= array2[secret]; // LS: Leak Secret
5 }
```

図 2.4: Spectre-PHT 脆弱性を含むコード片

2.3 一時実行攻撃

一時実行攻撃とは、CPU の投機的実行によって一時的に実行される命令の結果をマイクロアーキテクチャに痕跡を残すことを利用する攻撃法である。本来、CPU は誤った投機的実行が行われた場合、その結果はマイクロアーキテクチャの状態には反映されず、パイプラインはフラッシュされる。しかし、キャッシュなどの一部のマイクロアーキテクチャの状態はパフォーマンスの観点からそのまま維持される。これを利用して、攻撃者は投機的実行を誘発させ、マイクロアーキテクチャの状態を通じて、後から秘密情報などを復元することが可能である。

一時実行攻撃は 2018 年に Spectre 攻撃 [13] と Meltdown 攻撃 [15] が初めて明らかにされて以来、様々な CPU を標的とした、多数の新しい一時実行攻撃が発見されてきた。これらの攻撃は大きく分けて Spectre 型と Meltdown 型に分類される [6]。Spectre 型の攻撃 [11, 13, 14, 17] はデータフローまたは制御フローの予測ミスに続く一時的な命令を悪用する。一方で, Meltdown 型の攻撃 [15, 23, 26–28] は fault を発生させる命令に続く一時的な命令を悪用する。

一時実行攻撃は大きく分けて 3 つのフェーズで構成される。(TODO: 概要図)。まず攻撃者は分岐予測器やデータキャッシュの状態を設定し、マイクロアーキテクチャを目的の状態にする。次に、投機実行を引き起こすトリガー命令を実行する。これは、例外や分岐予測ミスなどにより、後続の命令が最終的に潰されるような命令である。CPU はトリガー命令が完了する前に後続の命令を一時的に実行する。この一時実行命令はマイクロアーキテクチャの covert channel の送信側として機能し、秘密に依存するメモリ位置を キャッシュにロードしたりする。トリガー命令の処理を終了すると、CPU は例外や分岐予測ミスを検知し、パイプラインをフラッシュし、アーキテクチャの状態をロールバックする。最後に攻撃者は、covert channel の受信側で、メモリアクセス時間の計測を行い、秘密情報を一時実行命令から推測するなどして、許可されていない一時的な命令の結果を復元する。

2.4 Spectre 攻撃と対策

Spectre 攻撃 [10, 13, 14, 17] はプロセッサの脆弱な投機実行を悪用することで、被害者プロセスの秘密情報を漏洩させる攻撃手法である。投機実行はプロセッサのパフォーマンス向上に大きく寄与しており、ほぼ全ての最新のプロセッサに実装されている最適化手法である。そのため、Spectre 攻撃は、特定のベンターに限らず、ほぼ全てのプロセッサが攻撃対象となる可能性がある。2018 年にプロセッサの Spectre 脆弱性が発見されて以降、多くの対処法が研究されているが、未だに Spectre 脆弱性を完全かつ効率的に排除する手法は確立されていない。本章では、本研究の対象であり Spectre 脆弱性の一種である Spectre-PHT [13] と、その脆弱性に対する代表的な防御手法について紹介する。

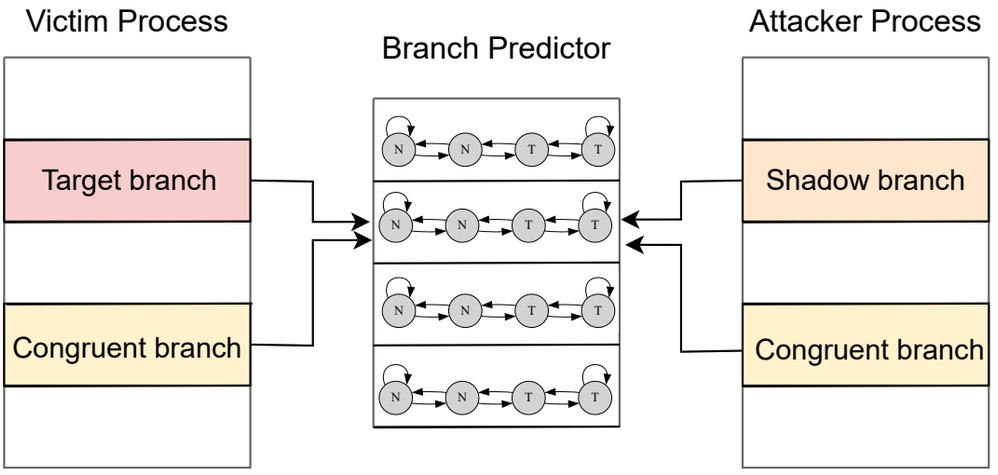

図 2.5: 分岐予測器のトレーニング

2.4.1 Spectre-PHT

Spectre-PHT(Spectre v1) [13] は Spectre 攻撃の一種であり、条件分岐命令の誤った予測によって引き起こされる投機実行による境界外メモリアクセスを利用する攻撃手法である。投機実行により条件分岐命令をバイパスするため、Bounds-Check-Bypass (BCB) 攻撃とも呼ばれる。

図 2.4 に示す、Spectre-PHT 脆弱性を含むコード辺を用いて、攻撃の具体的な手順を説明する。前提として、変数 `i` の値は外部から与えられ、攻撃者が操作可能であるとする。

攻撃者はまず、2 行目の `if` 文 (Victim branch, VB) の条件が True と予測されるように分岐予測器をトレーニングする必要がある。近年の CPU における分岐予測器は、通常、分岐命令の仮想アドレスをインデックスとして利用する [8, 13]。この特性を利用し、攻撃者は、Victim branch と同一の仮想アドレスを持つ分岐命令（Shadow branch）を用いて、自身のプロセス内で分岐予測器をトレーニングできる。また、仮想アドレスの一部のみが分岐予測器のインデックスとして使用される場合には、Victim branch と仮想アドレスの一部が一致する分岐命令（Congruent branch）を使用してトレーニングを行うことも可能である。これにより、攻撃者は図 2.5 に示すように、複数の経路を通じて分岐予測器をトレーニングできる [6]。図 2.4 のコード辺では、2 行目の条件分岐が True になるような入力を繰り返し与え、分岐予測器を True 側にトレーニングする。

次に、攻撃者は変数 `i` に `array1_size` 以上の値を与える。この場合、通常の実行では 2 行目の境界チェックによって、3 行目と 4 行目の処理は実行されずにプログラムは終了する。しかし、先述のトレーニングの結果、分岐予測器は分岐条件が真になると誤って予測するため、3,4 行目が投機実行される。この投機実行により、3 行目では攻撃者が操作可能な変数 `i` を使用して境界外アクセスが発生する (Read Secret, RS)。読み取られた値は、4 行目で `array2` へのアクセスのインデックスとして使用され、キャッシュの状態に変更が加えられる (Leak Secret, LS)。その後、CPU は分岐条件の評価結果から分岐予測が誤っていたことを知り、投機実行中のメモリアクセス結果を破棄するが、キャッシュの状態はパフォーマンスの観点からロールバックされない。

最後に、攻撃者は先述の Prime+Probe や Flush+Reload といった Side-Channel Attack を用いて、投機実行中のメモリアクセスで使用されたキャッシュラインを把握する。4 行目で読み取られたメモリアドレスは秘密情報に依存しており、使用されるキャッシュラインの位置は仮想アドレスによって決定するため、どのキャッシュラインが使用されていたかを把握することで、秘密情報を復元することが可

```
1 #include <x86intrin.h>
2 i = input();
3 if (i < array1_size) {
4   _mm_lfence();   // Add lfence
5   secret = array1[i];
6   tmp &= array2[secret];
7 }
```

図 2.6: Spectre-PHT 脆弱性を含むコード片に対し、lfence 命令による防御策を適用した例

能になる。

以上のことから、Spectre-PHT 脆弱性は、(i) 攻撃者が制御可能な分岐命令（VB)、(ii) 秘密情報を読み取る命令（RS)、(iii) 読み取った秘密情報をキャッシュ状態に反映させる命令（LS）の 3 つの命令から構成される。ただし、実際の攻撃では、VB に該当する命令の分岐予測から開始した投機実行が終了する前に、LS 及び RS に該当する命令の実行を完了する必要がある。これは、キャッシュ状態に秘密情報を反映させる前に投機実行が終了してしまうと、実行状態がロールバックされ、攻撃が成立しないためである。投機実行可能な命令数は CPU の ROB のサイズに制限されており (詳しくは 2.1 を参照)、投機実行可能な命令数の上限を投機ウィンドウ (SEW) という。このように、Spectre-PHT に対して脆弱なコード辺は Spectre Gadget と呼ばれる。以降では Spectre-PHT を単に Spectre と表記する。

2.4.2 直列化命令

Spectre 攻撃には投機実行が必要である。そのため、命令がその命令に至る制御フローが確定した場合にのみ実行されるようにすることで、Spectre 攻撃を防御することが可能である。このアプローチとして、元のプログラムに直列化命令を挿入して修正する方法が提案されている [13]。例えば x86 アーキテクチャの場合、lfence 命令を使用することが可能である。lfence 命令は、全ての先行するメモリロード命令が完了するまで、後続のメモリロード命令を投機実行させないようにする制御命令である。プログラム中の全ての条件分岐命令の分岐先に lfence 命令を挿入することで、それ以降のメモリロード命令の投機実行を抑制し、Spectre 攻撃を防御できる。

図 2.4 のコード片に対して lfence 命令を挿入して Spectre 攻撃に対して堅牢化したコード辺を図 2.6 に示す。4 行目に lfence 命令が挿入されることで、3 行目の分岐条件の評価が終了し分岐先が確定してから、後続の配列へのアクセスが行われるようになる。

しかし、CPU の分岐予測を無効化することで、大幅にパフォーマンスが低下することが知られている。既存研究 [30] では、元のプログラムの全ての条件分岐命令に対して lfence 命令を挿入した場合、プログラムの実行時間が最大 3.25 倍程度に増加することが報告されている。

2.4.3 Speculative Load Hardening

Spectre 攻撃に対する実行時オーバーヘッドが少ない防御策として、Carruth によって提案された Speculative Load Hardening(SLH) [7] がある。SLH はコンパイラベースの防御手法であり、分岐命令を使用せずに、メモリアクセス命令が有効な制御フローパス上で実行されているかを確認するコードを計装する。以降では、先行研究 [7] に示されているコード例を元に、SLH の概要を説明する。

図 2.7 と図 2.8 は、それぞれ SLH 適用前と後のコード辺である。ここで、関数 `leak` は投機実行された場合に引数のデータを攻撃者に漏洩させると仮定する。変数 `predicate_state` は、現在の実行が分

```
1  void leak(int data);
2  void example(int* pointer1, int* pointer2) {
3    if (condition) {
4      // ... lots of code ...
5      leak(*pointer1);
6    } else {
7      // ... more code ...
8      leak(*pointer2);
9    }
10 }
```

図 2.7: SLH 適用前の Spectre Gadget

```
1  uintptr_t all_ones_mask = std::numerical_limits<uintptr_t>::max();
2  uintptr_t all_zeros_mask = 0;
3  void leak(int data);
4  void example(int* pointer1, int* pointer2) {
5    uintptr_t predicate_state = all_ones_mask;
6    if (condition) {
7      predicate_state = !condition ? all_zeros_mask : predicate_state;
8      // ... lots of code ...
9      pointer1 &= predicate_state;
10     leak(*pointer1);
11   } else {
12     predicate_state = condition ? all_zeros_mask : predicate_state;
13     // ... more code ...
14     int value2 = *pointer2 & predicate_state;
15     leak(value2);
16   }
17 }
```

図 2.8: SLH 適用後のコード辺

岐予測ミスによる分岐先であるかを表しており、7 行目と 12 行目において、条件 `condition` が満たされない場合は `all_zeros_mask`(全てのビットが 0 の値)、満たしている場合は `all_ones_mask`(全てのビットが 1 の値) が代入される。重要な点として、7 行目の三項演算子は分岐命令が使用されない形で機械語に変換される必要がある。これにより、7 行目の条件 `condition` が投機実行によってバイパスされないことを保証する。x86 アーキテクチャでは、cmov 命令を用いることで実現可能である [7]。

分岐予測が正しい場合は、varpredicate_state には `all_ones_mask` が格納される。そのため、9 行目及び 13 行目で論理積が取られても、`pointer1` 及び `pointer2` の値はそのまま保持される。一方、分岐予測が誤っている場合、varpredicate_state には `all_zeros_mask` が格納されている。これにより、論理積を取ることで、`pointer1` と `pointer2` の値は 0 となる。その結果、関数 `leak` が実行されても攻撃者が意図したデータは漏洩しない。

lfence 命令を用いた防御策では、lfence 命令以降に続くすべてのメモリロード命令の投機実行が抑制

される。このため、Spectre 攻撃に関与しない安全なメモリロード命令であっても、投機実行が制限されてしまう。また、分岐予測が正しく行われた場合でも、lfence 命令以降の投機実行が抑制されるため、パフォーマンスが大幅に低下する。一方、SLH は、投機実行による漏洩のリスクがある危険なメモリロード命令に対してのみ投機実行を抑制する。そのため、投機実行可能なメモリロード命令を増やすことができ、lfence 命令と比較して小さい実行時間オーバーヘッドで Spectre 攻撃を防御することができる。しかし、それでも全てのメモリロード命令を SLH で強化する場合、大規模なアプリケーションでは実行時オーバーヘッドが 30% 程度になることが報告されている [7]。

2.5 Spectre Gadget の検出

Spectre 攻撃に対してプログラムを堅牢化するために、lfence 命令や SLH を全ての条件分岐命令やメモリロード命令に適用する場合、大きな実行時オーバーヘッドが発生する。そこで、プログラム内の潜在的な Spectre gadget を見つけ、これらの gaget のみに防御策を適用して、実行時オーバーヘッドを軽減する方法が提案されている。

Spectre Gadget を検出するためのプログラム解析手法は大きく分けて、静的解析 [2, 9, 29, 30] による手法と、動的解析 [12, 19, 20] による手法の 2 種類に分類される。投機実行はハードウェアによる機能であるため、これらの解析手法の大半がソフトウェアレベルで投機実行をシミュレートすることで Spectre gadget を検出している。以降では、それぞれの手法がどのように投機実行をシミュレートし、Spectre gadget を検出しているかを説明する。その代表例として、静的解析による検出手法として KLEESpectre [29] を、動的解析による検出手法として SpecFuzz [19] を紹介する。これらの既存手法は本研究における実装の基盤にもなっているため詳しく説明する。

2.5.1 KLEESpectre

KLEESpectre [29] は記号実行により、プログラム中の Spectre gadget を検出するツールである。記号実行とは、プログラムに具体的な値ではなく、記号的な変数を入力として与えることで、プログラムの全ての実行パスを網羅的に解析する静的手法である。入力を記号化することで、プログラムが異なる入力で取る可能性のある複数のパスを同時に探索することができる。記号実行は記号実行エンジンによって行われ、探索された制御フローパスごとに、(i) そのパスに沿って実行された分岐によって満たされる条件を記述するパス制約、および (ii) 各変数を記号式または具体的な値にマップする記号的なメモリ状態を保持することで現在の実行状態を管理する [4]。

KLEESpectre は記号実行エンジンである KLEE [5] を Spectre gadget の検出用に拡張したものであり、単純なパターンマッチングによる手法 [2] と比較して高い精度で Spectre gadget を検出することができる。

まず、[29] に挙げられるコード辺を用いて、KLEESpectre がどのように分岐予測による投機実行をシミュレートしているかを説明する。図 2.9 は典型的な Spectre 脆弱性を含んでいるコード辺である。分岐 `b1` が誤って分岐予測された場合、`x` の値は `SIZE` 以上となるので、8 行目で境界外アクセスが発生し、秘密情報が `temp` に読み取られる可能性がある。その後、9 行目で `temp` が `array2` へのアクセスのインデックスとして使用されることで、秘密情報がキャッシュの状態として漏洩する。

通常の記号実行の場合、条件分岐命令に遭遇すると、分岐条件が満たし、Taken 側の基本ブロックに進んだ記号状態と、分岐条件を満たさず、Not taken 側の基本ブロックに進んだ記号状態の 2 つの状態が新しく生成される。しかし、KLEESpectre では投機実行をシミュレートするため、上記の 2 つの状態に加え、分岐条件が満たし、Not taken 側の基本ブロックに進んだ記号状態と、分岐条件を満たさず、Taken 側の基本ブロックに進んだ記号状態の 2 つの状態も生成される。

```
uint32_t SIZE = 16;
uint8_t array1[16], array2[256*64], array3[16];

uint8_t foo(uint32_t x) {
  uint8_t temp = 0;
  if(x < size) {    // b1
    // A
    temp = array1[x];
    temp |= array2[temp];
    if(x <= 8) {  // b2
      // B
      temp |= array2[8];
    }
  }
  // C
  temp |= array3[8]
  return temp;
}
```

図 2.9: KLEESpectre におけるコード例

例えば、KLEESpectre が図 2.9 の分岐 `b1` に遭遇した場合、以下の 4 つの状態が新しく生成される。

(1) `x < SIZE` を満たし、分岐 `b1` が正しく分岐予測された状態
(2) `x < SIZE` を満たさず、分岐 `b1` が正しく分岐予測された状態
(3) `x < SIZE` を満たし、分岐 `b1` が誤って分岐予測された状態
(4) `x < SIZE` を満たさず、分岐 `b1` が誤って分岐予測された状態

(1) の場合、KLEESpectre は現在の記号状態のパス制約に `x < SIZE` を追加し、基本ブロック `A` に移動し、実行を続ける。(2) の場合も同様に、現在の記号状態のパス制約に `x >= SIZE` を追加し、基本ブロック `C` に移動し、実行を続ける。(3) の場合、KLEESpectre は現在の記号状態のパス制約に `x < SIZE` を追加するが、分岐予測ミスにより、基本ブロック `C` に移動し、実行を続ける。(4) の場合も同様に、現在の記号状態のパス制約に `x >= SIZE` を追加するが、分岐予測ミスにより、基本ブロック `A` に移動し、実行を続ける。このように KLEESpectre は分岐予測ミスによる投機的なパスも考慮して記号実行を行うように KLEE を拡張している。図 2.10 において、投機的なパスも含めた図 2.9 の制御フローグラフを示す。

図 2.10 では、実線が直前の分岐命令が正しく分岐予測された場合のパスを、点線が直前の分岐命令が誤って分岐予測された場合の投機的パスを表している。緑色の基本ブロックは通常の実行パスにおける基本ブロックを、赤色の基本ブロックは投機的なパスにおける基本ブロックを表している。KLEESpectre は全ての分岐命令が攻撃者によって訓練されていると仮定し、投機実行のシミュレートを行う。このシミュレーションにおいて、投機的な状態が以下のいずれかの条件を満たした場合、CPU における投機実行の終了動作を再現するためにその記号状態を破棄する。

- 投機的なパス上で実行した命令数が投機ウィンドウの制限に達した場合
- 直列化命令に遭遇した場合

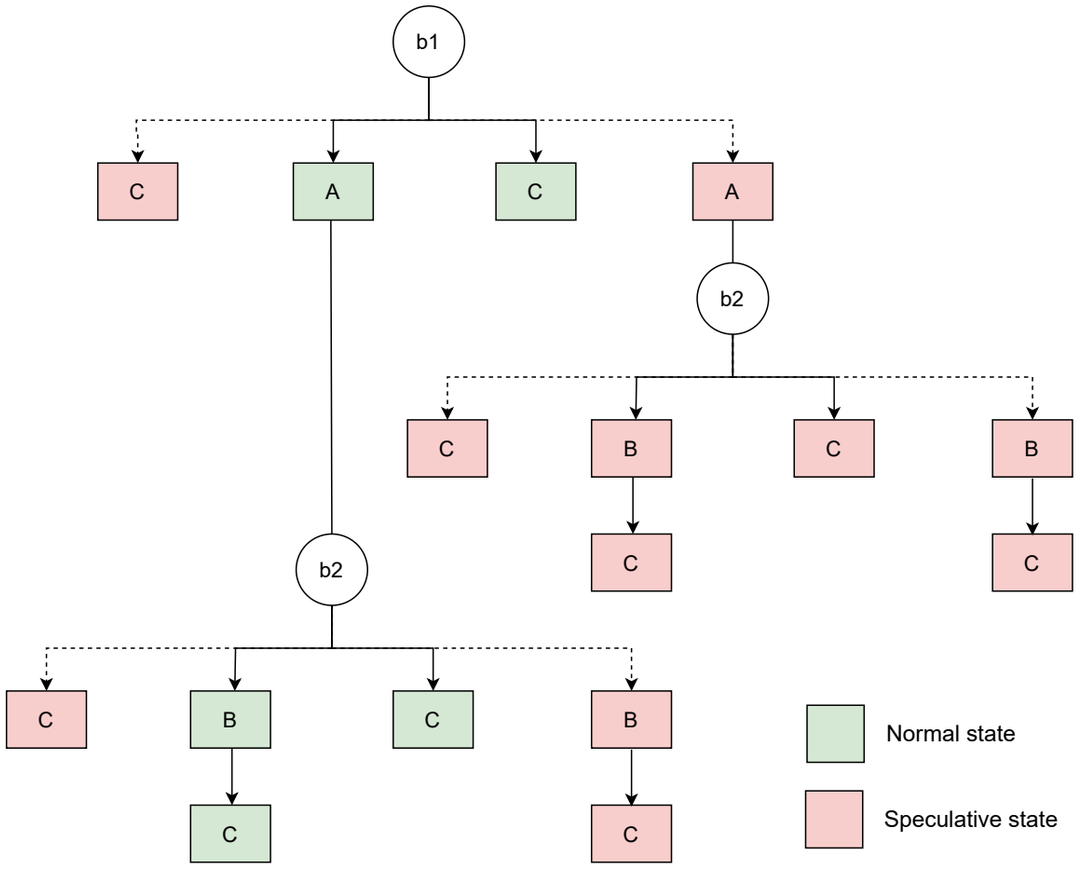

図 2.10: 投機的なパスを含んだ制御フローグラフ

- 例外が発生した場合

また、本研究のテーマと関連する重要な点として、KLEESpectre では投機実行をネストしてシミュレートする場合がある。図 2.9 では、分岐 `b1` の分岐予測ミスによる投機実行中に分岐 `b2` に遭遇した場合、KLEESpectre は重ねて分岐 `b2` の分岐予測ミスをシミュレートする。このように、KLEESpectre では、ある分岐命令の分岐予測ミスから開始された投機実行中に、別の分岐命令に遭遇した場合、重ねて投機実行をシミュレートする。実際の CPU においても、このようなネストされた投機実行は可能であり [18]、KLEESpectre は正しく CPU の投機実行をシミュレートしていると言える。

次に、KLEESpectre がどのように Spectre Gadget を検出するかについて説明する。まず、投機的に実行されたパスにおけるメモリアクセスを監視し、それらが秘密情報を参照している場合、その命令を RS（Read Secret）として記録する。KLEESpectre では、投機的パスにおいて攻撃者が操作可能な値（記号変数）をアドレスとして使用した境界外のメモリアクセスは、すべて秘密情報を参照していると仮定し、保守的な解析を行っている。境界外のメモリアクセスは KLEE に組み込まれているチェック機構を利用して、識別している。次に、秘密情報に依存する値を用いてメモリアクセスが行われた場合、その命令を LS（Leak Secret）として記録する。LS が検出されると、直前に分岐予測ミスが発生した分岐命令と、RS および LS に該当する命令のセットをまとめて、Spectre Gadget として記録する。

KLEESpectre は記号実行を活用することで、単純な静的解析手法や動的解析手法に比べて、プログラムの網羅的な解析が可能である。これにより、より高い精度で Spectre gadget を検出できる。しかし、大規模なコードや探索すべきパスが多い複雑なプログラムに対しては、スケールしないという課題がある。

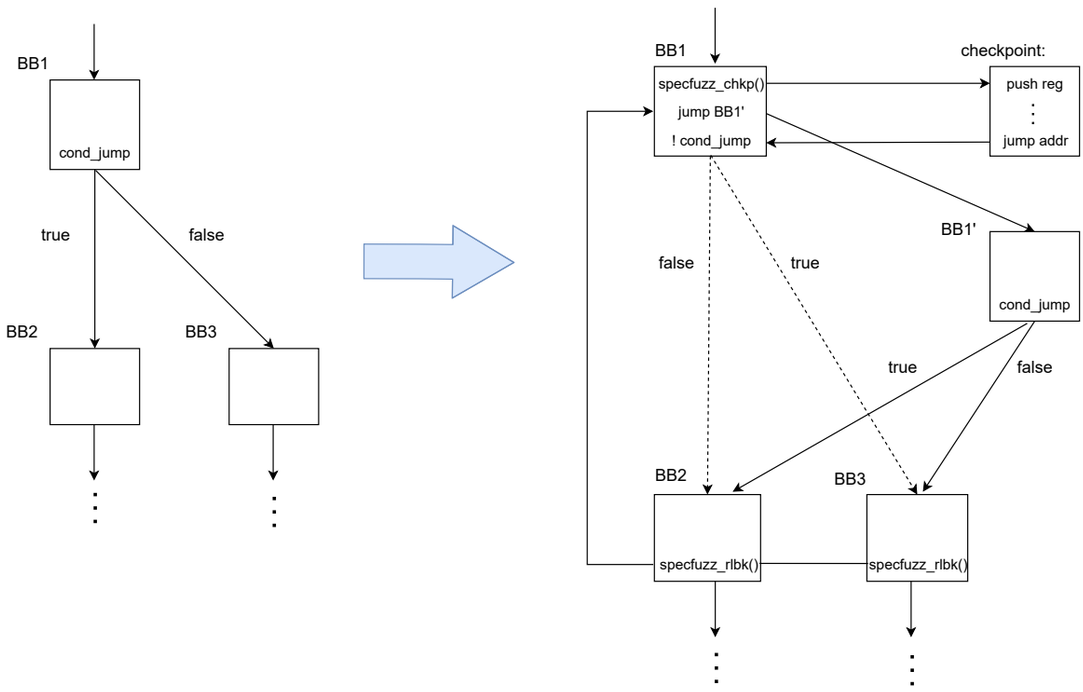

図 2.11: SpecFuzz による計装前と計装後の制御フローグラフの概略

2.5.2 SpecFuzz

SpecFuzz [19] は、ファジングを利用してプログラム内の Spectre Gadget を検出するためのツールである。ファジングとは、プログラムにランダムに生成された入力を与えることで、バグや脆弱性を検出する動的解析手法である。ファザーは、文法に基づいてゼロからランダムな入力を生成するか、既存の入力コーパスを変更して新たな入力を作成する。これらの生成された入力をプログラムに与え、その挙動を監視することで、未知のバグや脆弱性を特定する。また、ファジングの重要なパラメーターの 1 つにカバレッジがある。これは、ファジング中にプログラムがどの程度広範囲にテストされたかを示している。一般的には、ファジング中に少なくとも 1 回実行された制御フローグラフのエッジ数とプログラム内のエッジの合計数の比率として定義される。当然カバレッジが低いと検出対象の見逃しが発生するため、カバレッジを向上させることが多くのファザーにとって重要である。

まず、SpecFuzz がどのように分岐予測による投機実行をシミュレートしているかを説明する。SpecFuzz は、テスト対象のプログラムに対して x86 アーキテクチャ用の LLVM コンパイラバックエンドパスを利用し、投機実行のシミュレーションに必要なコードを計装する。図 2.11 に計装前のプログラムの制御フローグラフと SpecFuzz による計装後の制御フローグラフの概略図を示す。点線が分岐予測ミスによるパス、実線が通常の実行によるパスを表している。また、KLEESpectre と同様に、全ての分岐命令が攻撃者によってトレーニングされていると仮定し、投機実行をシミュレートする。

SpecFuzz では、まず全ての条件分岐命令の直前にチェックポイントを配置する関数 (`specfuzz_chkp`) の呼び出しを計装する。この関数は、投機実行のシミュレーション開始地点を示し、シミュレーション終了後に通常の実行パスへ戻るため、現在の CPU 状態を保持する役割を持つ。具体的には、以下の情報をスナップショットとして取得し、メモリに保存する。

- レジスタ値（GPR、フラグ、SIMD、浮動小数点レジスタなど)
- ロールバック先のアドレス

- メタデータ (スタックポインタ、シミュレーション中に実行した命令数など)

次に、全ての分岐命令 (`cond_jump`) を、その分岐条件を反転させた命令 (`!cond_jump`) に置き換える。この操作により、分岐命令に遭遇するたびに通常の実行パスとは逆方向に処理が進み、投機実行のシミュレーションが開始する。さらに、メモリを変更する全ての命令 (mov, push, call など）の直前には、変更対象のアドレスとその直前の値を記録するコードを挿入する。これにより、投機実行のシミュレーション中に行われた全てのメモリの変更がログに記録され、ロールバック時にシミュレーション開始時点の状態に戻すことが可能になる。

投機実行のシミュレーションは、Machine IR (MIR) レベルで実行された命令数が投機ウィンドウによる上限に達するか、直列化命令に遭遇した場合に終了する。各基本ブロックの最後では、シミュレーション中に実行された命令数が投機ウィンドウに達しているかをチェックし、上限に達していない場合、シミュレーションを継続する。上限に達していた場合、シミュレーションを終了するため、ロールバック用の関数 (`specfuzz_rlbk`) が呼び出され、直前のチェックポイント地点にシャンプし、スナップショットに基づいてレジスタやメモリの状態を復元した後、正しい実行パスを再開する。

次に、SpecFuzz がどのように Spectre Gadget を検出するかについて説明する。SpecFuzz は、投機実行のシミュレーション中に境界外アクセスが検出された場合、その直前に分岐予測ミスが発生した分岐命令と、境界外アクセスが行われた命令を合わせて Spectre gadget として報告する。境界外アクセスの検出には、AddressSanitizer（ASan）[21] が使用される。

この手法は境界外アクセスが攻撃者の操作している値に依存しているかや、境界外アクセスで読み取った値がキャッシュへ転送されるか (LS が存在するか) といったことは考慮されておらず、単純な検出手法である。そのため、実際には攻撃に利用できない gadget も多数検出される可能性がある [20]。しかし、単純化された検出機構により他のファジングベースの Spectre gadget の検出手法 [12, 20] と比較して、より高いファジングスループットを実現できる。

SpecFuzz はファジングを活用することで、記号実行を用いた手法と比較して大規模なプログラムに対してもスケールする。しかし、ファジングのカバレッジ不足により、一部の Spectre gadget を見逃す可能性がある。また、検出機構が単純化されているため、実際には攻撃に利用できない gadget が誤って検出される可能性もある。

第 3 章

問題設定

本論文では、スケーラビリティと精度を両立する Spectre ガジェットの検出手法を提案する。Spectre ガジェットを正確かつ効率的に検出することは、最低限の実行時オーバーヘッドでプログラムを Spectre 攻撃に対して堅牢化するために不可欠である。以降では、まず既存の Spectre ガジェット検出手法が抱える問題点について述べる。その後、その問題点を示す Motivating Example を提示し、本研究の目的を明確化する。

3.1 既存手法の問題点

3.1.1 記号実行

記号実行を用いた Spectre gadget の検出手法 [9, 29] の問題点は、大規模なプログラムに対してスケールしないという点である。これは一般的な記号実行による解析に共通する問題でもあるが、Spectre gadget の検出ツールは通常のパスに加えて、投機的なパスも考慮する必要があるため、これはより顕著な問題となる。この問題の主な原因はパス爆発とストア状態の爆発として知られている [4]。

パス爆発とは、プログラム内の分岐数に比例して探索すべき実行パスが指数関数的に増加する現象を指す。記号実行エンジンは、分岐箇所に遭遇すると現在の状態を分岐方向ごとに分割し、新たな状態を生成する。このため、分岐が多いプログラムでは実行パスの数が急激に増加し、解析のための実行時間やメモリ使用量が膨大となる。結果として、すべての実行パスを網羅的に解析することが現実的に不可能となる場合がある。

パス爆発の主な原因の一つは、ループ構造に起因するものである。ループは分岐命令として扱われるため、各反復ごとに分岐方向に対応する新しい状態が生成される。さらに、図 3.1 に示すように、ループ条件に記号化された変数が含まれる場合、ループの反復回数が具体的に決定できず、最大のループ回数を想定して解析が行われる可能性がある。一般的なツールでは、ループの解析を限られた回数に制限しているが、この場合、検出対象を見逃す可能性もある。

```
1 int i;
2 symbolic_var(i); // Symbolize variable i
3 while(i > 0) {
4   ...
5   i--;
6 }
```

図 3.1: 記号実行におけるパス爆発

```
1 size_t i, j;
2 int temp;
3 symbolic_var(i); // Symbolize variable i
4 symbolic_var(j); // Symbolize variable j
5 int array[5] = {0};
6 if(i < 5 && j < 5) {
7   array[i] = 1;
8   temp = array[j];
9 }
```

図 3.2: 記号実行におけるストア状態の爆発

ストア状態の爆発とは、記号実行において抽象的に扱われるメモリ状態の数が急激に増加する現象を指す。記号実行エンジンは、プログラムのメモリ操作を正確にモデル化するために、各メモリアドレスを具体的な値または記号式に関連付けたストア状態というデータ構造を保持する。そのため、記号化されたアドレスに対して読み取りや書き込みが行われる場合、操作の結果として生じる可能性のある全てのストア状態を考慮する必要がある。その結果、新しいストア状態が次々に生成され、ストア状態の数が急激に増加する。この現象は、大規模なプログラムや複雑なメモリアクセスパターンを持つコードにおいて特に顕著である。

図 3.2 に示すコード例を用いて、ストア状態の爆発について具体的に説明する。このコードでは、変数 i と j が記号化されている。まず、7 行目の配列への書き込みに注目する。この箇所では、i が記号化されているため、array[0] から array[4] のいずれかの要素が 1 に書き換えられる可能性がある。その結果、これら 5 つのケースを全て考慮するため、7 行目の実行後に現在の状態から 5 つの新しい状態が分岐される。同様に、8 行目の配列への読み取りにおいても、j が記号化されているため、array[0] から array[4] のいずれかの要素が読み取られ、変数 temp に格納される。この場合も、5 つのパターンを考慮するため、8 行目の実行後にさらに 5 つの新しい状態が分岐し、最終的には最初の状態から新しく 25 個の状態に分岐することになる。

この例では、変数 i,j の範囲が制限されていたため、メモリ操作が参照する可能性のあるアドレスのセットは比較的小さかったが、一般的には、記号化されたアドレスはメモリ内の任意のアドレスを参照する可能性があるため、ストア状態の数が爆発的に増加する可能性がある。

パス爆発とストア状態の爆発の問題は、記号実行を用いて Spectre gadget を検出する場合により顕著になる。これは、分岐予測ミスによる投機的なパスを考慮することで、通常の記号実行に比べて探索すべきパス数が大幅に増加するからである。通常の実行パス上の状態と異なり、投機的なパス上の状態は投機ウィンドウの上限に達した時点で破棄されるため最後まで残ることはない。しかし、それでも解析時間や最大メモリ使用量に大きな影響を与えるため、大規模なプログラムや複雑な制御フローを持つプログラムに対しては特にスケールしない。

3.1.2 ファジング

ファジングを用いた Spectre gadget の検出手法における主な問題点は、カバレッジ不足による Gadget の見逃しが発生する可能性がある点である。ファジングは記号実行とは異なり、生成された入力によって実行されたパス上に存在する gadget しか検出できない。この制約は、一般的なファジングツールに共通する問題でもある。

さらに、ファザーが脆弱性を引き起こす入力を生成しない場合、Spectre gadget を検出することはで

きない。例えば、図 2.4 に示すコードを考える。ここで、ファザーが条件 `i < array1_size` を満たす入力 `i` を生成した場合、3 行目および 4 行目は通常実行される。そのため当然、3 行目で境界外アクセスが発生しないので、脆弱性は検出されない。一方、`i < array1_size` を満たさない入力 `i` を生成することで、3 行目及び 4 行目が投機実行がシミュレートされ、3 行目の境界外アクセスが引き起こされ、初めて脆弱性を検出できる。このように、ファジングによる Spectre gadget の検出は、通常の実行パスだけでなく、投機的なパスも漏れなくテストすることが必要である。

3.2 MotivatingExample

図 3.3 は、記号実行を用いた Spectre gadget の検出において、投機的な状態数が爆発する具体例を示している。この図には、3 つの Spectre gadget が含まれている。1 つ目は、18 行目の分岐予測がミスされた後に実行される 9 行目の命令である。2 つ目は、9 行目と 18 行目の両方の分岐予測がミスされた後に実行される 10 行目の命令である。そして 3 つ目は、18 行目、9 行目、11 行目の全ての分岐予測がミスされた後に実行される 12 行目の命令である。先述の通り、CPU はネストされた投機実行が可能であり、既存の記号実行による手法 [9, 29] では、これらの gadget を漏れなく検出するため、ネストされた分岐予測ミスをシミュレートする必要がある。そのため、投機ウィンドウの上限に達するまで、分岐命令に遭遇するたびに以下の 4 つの新たな状態が生成され、状態数の爆発に大きく寄与していると考えられる。

- 分岐条件を満たし、Taken 側へ遷移した状態
- 分岐条件を満たさず、Not taken 側へ遷移した状態
- 分岐条件を満たさず、分岐予測ミスにより Taken 側へ遷移した状態
- 分岐条件を満たし、分岐予測ミスにより Not taken 側へ遷移した状態

しかし、既存研究 [19] では、現実世界の検体において、10 行目や 12 行目のような、ネストされた分岐予測ミスによりトリガーされる gadget は少ないことが報告されている。この直感的な理由として、多くのメモリアクセス命令は、単一の境界チェック条件で保護されており、10 行目や 12 行目のような複数の境界チェックで保護されているメモリアクセス命令は稀であるからと考えられる。また、複数の分岐命令を訓練することは、攻撃者にとって非常に困難であるため、このような gadget が実際の攻撃で利用される可能性は低いと考えられる。

既存の記号実行による手法では、gadget が検出される可能性の低い (あるいは、検出しても攻撃に利用されにくい) ネストされた分岐予測ミスにより到達する投機的な状態まで探索しているため、スケーラビリティが低下していると考える。

表 3.1 に図 3.3 を KLEESpectre で解析を行った結果を示す。実験環境は以降の説の通りである、投機ウィンドウのサイズは LLVM IRb レベルで 200 命令に設定した。表 3.1 からわかる通り、小規模な検体でありながら、通常実行において探索した状態数 (State) に比べて、非常に多くの投機的な状態数 (Speculative states) を探索している。これにより、通常の記号実行による解析と比較して、解析時間と最大メモリ使用量が大幅に増加していると考えられる。

表 3.1: KLEESpectre による MotivatingExample の解析結果

Detected gadgets	**States**	**Speculative states**	**Time of analysis (h:m:s)**	**Maximum Memory Usage (KB)**
1	1	1	1	1

```
#include <stdint.h>
#include <stdlib.h>
#define ARRAY1_SIZE 16
uint8_t array1[16] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
uint8_t array2[256 * 512];
uint8_t temp = 0;

void var(size_t index) {
    if (index < ARRAY1_SIZE / 2) {   // br2
        // C
        temp &= array2[array1[index] * 512];
        if (index < ARRAY1_SIZE / 4) {  // br3
            // D
            temp &= array2[array1[index] * 512];
        }
    }
    // E
}

void foo(size_t index) {
  while (index < ARRAY1_SIZE) {  // br1
    // A
    temp &= array2[array1[index] * 512];
    var(index);
    ++index;
  }
    // B
}

int main() {
  size_t num = input();
  foo(num);
  return 0;
}
```

図 3.3: Motivating Example

第 4 章

提案手法

本研究では、記号実行とファジングの解析手法を組み合わせることで、スケーラビリティと精度の両立を目指す Spectre gadget 検出手法を提案する。記号実行とファジングにはそれぞれ利点と欠点が存在するが、これらを組み合わせることで、両者の欠点を補完し、スケーラビリティと精度を両立できるのではないかと考える。提案手法は、記号実行フェーズとファジングフェーズの 2 つのフェーズで構成され、それぞれのフェーズで検出された gadget を合わせて、最終的な検出結果として報告する。記号実行フェーズでは、ネストされた分岐予測を制限することで、投機的状態の探索範囲を縮小し、解析のスケーラビリティを向上させる。ファジングフェーズでは、記号実行フェーズで探索されなかった投機的な状態を集中的にテストすることで、記号実行で見逃された gadget を効率的に検出することを目指す。

以下では、記号実行フェーズとファジングフェーズのそれぞれについて、提案手法の詳細を説明する。

4.1 記号実行フェーズ

4.1.1 ネストされた分岐予測の制限

記号実行フェーズでは、ネストされた分岐予測を制限することでスケーラビリティの向上を図る。提案手法の記号実行フェーズがどのように投機実行をシミュレートするかについて、図 3.3 の制御フローグラフの一部である図 4.1 を用いて説明する。点線が分岐予測ミス、実線が通常の実行パスを表す。

まず、既存手法と同様に、解析が分岐 `b1` に到達すると、投機実行のシミュレーションが開始され、以下の 4 つの状態に分岐する。

1. 分岐 `b1` の条件を満たし、A に遷移した状態
2. 分岐 `b1` の条件を満たさず、B に遷移した状態
3. 分岐 `b1` の条件を満たさず、分岐予測ミスにより A に遷移した状態
4. 分岐 `b1` の条件を満たし、分岐予測ミスにより B に遷移した状態

これらのうち、3 番目の状態のみが 23 行目の脆弱性を引き起こし、Spectre gadget として検出される。この状態から解析を進めていくと、次に分岐 `b2` に遭遇する。既存手法では、ネストされた投機実行をシミュレートするため、`b2` でも分岐予測ミスを含めた 4 つの状態に分岐する。一方、提案手法ではネストされた分岐予測を制限し、`b2` における分岐予測ミスによる投機的な状態は探索しない。そのため、この状態において、以降は、投機ウィンドウの上限に達するか、直列化命令に遭遇するまで、正しい分岐方向の状態のみを探索し続ける。

その結果、11 行目と 14 行目の脆弱性を引き起こす gadget が見逃されてしまう。しかし、先述の通り、ネストされた分岐予測ミスにより引き起こされるされる gadget は稀であるため、影響は少ないと考える。また、このようなネストされた分岐予測ミスにより到達する投機的な状態は、後述するファジ

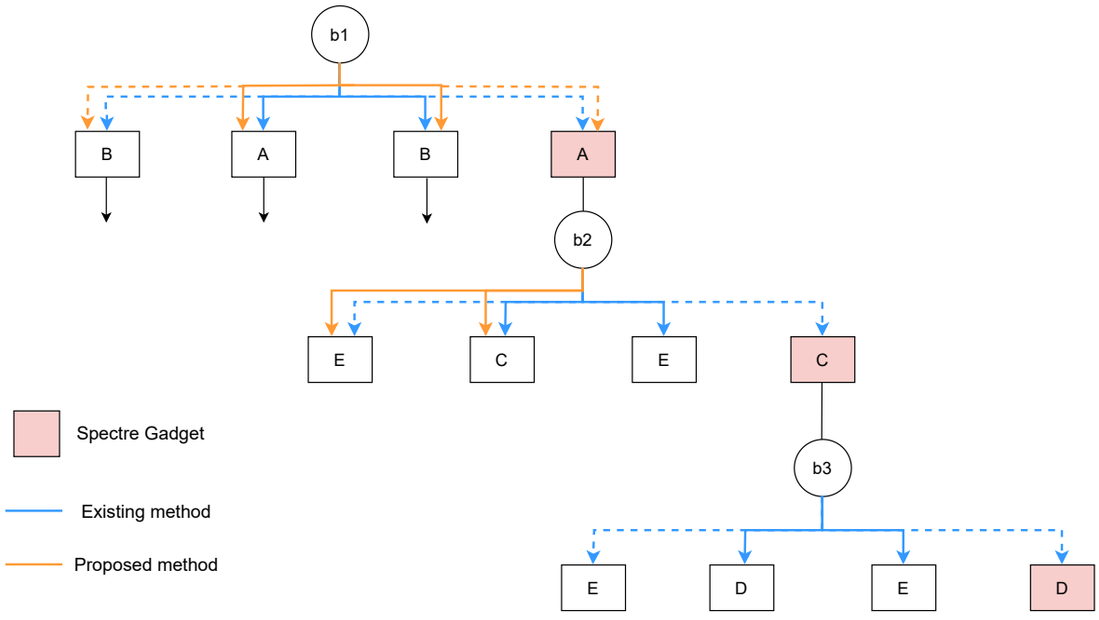

図 4.1: ネストされた分岐予測ミスの制限

ングフェーズで探索を行うことで、False nagative を最小限に抑え、精度を担保できると考える。

4.1.2 不要な分岐予測ミスの特定

記号実行で探索されなかったネストされた分岐予測ミスにより到達する投機的な状態はファジングフェーズで探索を行う。この際、記号実行で探索した投機的な状態を再度、ファザーで探索するのは非効率的である。そこで、記号実行の過程で、分岐予測ミスをシミュレートしても、シミュレーションが終了するまでに別の分岐命令に到達できない分岐方向を特定する。その情報をファザーに提供し、それらの分岐方向への分岐予測ミスのシミュレーションを抑制することで、ファジングのスループットを向上させる。

図**??**に示すコード例を用いて具体的に説明する。

4.1.3 目的状態までの実行トレースの取得

4.2 ファジングフェーズ

4.2.1 不要な分岐予測の抑止

4.2.2 スコアによるシードのスケジューリング

第 5 章

実装

5.1 記号実行フェーズ

5.2 ファジングフェーズ

第 6 章

予備実験

6.1 実験目的

RQ

6.2 実験準備

ファザーの初期シードの生成方法どの変数を記号化するかコンパイル時間は含まない検体ごとの埋め込む脆弱性の数は検体の規模に応じて、埋め込む

6.2.1 比較対象

6.2.2 検体

6.2.3 評価指標

ドライバも含めたサイズを記載

6.2.4 実験環境

6.3 実験結果

6.3.1 解析全体の評価

6.3.2 記号実行フェーズの評価

6.3.3 ファジングフェーズの評価

6.4 議論

第 7 章

今後の展望

第 8 章

関連研究

8.1 ソフトウェアによる防御

8.2 Spectre ガジェットの検出

8.3 ハードウェアによる防御

第 9 章

結論

謝辞

参考文献

[1] Refined speculative execution terminology. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html.

[2] Spectre variant 1 scanning tool, 2018. https://access.redhat.com/blogs/766093/posts/3510331.

[3] Onur Acıiçmez and Çetin Kaya Koç. Trace-driven cache attacks on aes (short paper). In Information and Communications Security: 8th International Conference, ICICS 2006, Raleigh, NC, USA, December 4-7, 2006. Proceedings 8, pages 112–121. Springer, 2006.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D' elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 51(3):1–39, 2018.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, volume 8, pages 209–224, 2008.

[6] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In 28th USENIX Security Symposium (USENIX Security 19), pages 249–266, 2019.

[7] Chandler Carruth. Speculative load hardening, 2018. https://llvm.org/docs/SpeculativeLoadHardening.html.

[8] Agner Fog. The microarchitecture of intel, amd and via cpus: An optimization guide for assembly programmers and compiler makers. Software optimization resources, 2016.

[9] Marco Guarnieri, Boris Köpf, José F Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1–19. IEEE, 2020.

[10] J. Horn. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. Accessed: December, 2023.

[11] Jann Horn. speculative execution, variant 4: speculative store bypass. 2018.

[12] Brian Johannesmeyer, Jakob Koschel, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Kasper: Scanning for generalized transient execution gadgets in the linux kernel. In NDSS, volume 1, page 12, 2022.

[13] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In 2019 IEEE Symposium on Security and Privacy (SP), pages 1–19, 2019.

[14] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-

Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In 12th USENIX Workshop on Offensive Technologies (WOOT 18), Baltimore, MD, August 2018. USENIX Association.

[15] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In 27th USENIX Security Symposium (USENIX Security 18), pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[16] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In 2015 IEEE symposium on security and privacy, pages 605–622. IEEE, 2015.

[17] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. CCS '18, page 2109–2122, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Speculator: a tool to analyze speculative execution attacks and mitigations. In Proceedings of the 35th Annual Computer Security Applications Conference, pages 747–761, 2019.

[19] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. In 29th USENIX Security Symposium (USENIX Security 20), pages 1481–1498, 2020.

[20] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In NDSS, pages 1–14, 2021.

[21] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In 2012 USENIX annual technical conference (USENIX ATC 12), pages 309–318, 2012.

[22] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on {SSH}. In 10th USENIX Security Symposium (USENIX Security 01), 2001.

[23] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. arXiv preprint arXiv:1806.07480, 2018.

[24] Jakub Szefer. Survey of microarchitectural side and covert channels, attacks, and defenses. Journal of Hardware and Systems Security, 3(3):219–234, 2019.

[25] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. Journal of Cryptology, 23:37–71, 2010.

[26] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In 27th USENIX Security Symposium (USENIX Security 18), pages 991–1008, 2018.

[27] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. Lvi: Hijacking transient execution through microarchitectural load value injection. In 2020 IEEE Symposium on Security and Privacy (SP), pages 54–72. IEEE, 2020.

[28] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In 2019

IEEE Symposium on Security and Privacy (SP), pages 88–105. IEEE, 2019.

[29] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. ACM Transactions on Software Engineering and Methodology (TOSEM), 29(3):1–31, 2020.

[30] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. arXiv preprint arXiv:1807.05843, 2018.

[31] Yuval Yarom and Katrina Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. In 23rd USENIX security symposium (USENIX security 14), pages 719–732, 2014.